Programming Distributed Memory Systems with MPI

Tim Mattson
Intel Labs.

With content from Kathy Yelick, Jim Demmel, Kurt Keutzer (CS194) and others in the UCB EECS community. www.cs.berkeley.edu/~yelick/cs194f07,
Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments
Tracking Supercomputers: Top500

- Top500: a list of the 500 fastest computers in the world (www.top500.org)
- Computers ranked by solution to the MPLinpack benchmark:
  - Solve $Ax=b$ problem for any order of $A$
- List released twice per year: in June and November

Current number 1 (June 2013) 33.9 PFLOPS
Tianhe-2, NUDT, Intel Ivy Bridge + Xeon Phi
16.3 PFLOPS, >1.5 million cores

Source: http://s.top500.org/static/lists/2013/06/TOP500_201306_Poster.pdf
The birth of Supercomputing

- The CRAY-1A:
  - 2.5-nanosecond clock,
  - 64 vector registers,
  - 1 million 64-bit words of high-speed memory.
  - Peak speed:
    - 80 MFLOPS scalar.
    - 250 MFLOPS vector (but this was VERY hard to achieve)

- Cray software … by 1978
  - Cray Operating System (COS),
  - the first automatically vectorizing Fortran compiler (CFT),
  - Cray Assembler Language (CAL) were introduced.

On July 11, 1977, the CRAY-1A, serial number 3, was delivered to NCAR. The system cost was $8.86 million ($7.9 million plus $1 million for the disks).

http://www.cisl.ucar.edu/computers/gallery/cray/cray1.jsp
History of Supercomputing:

- Large mainframes that operated on vectors of data
- Custom built, highly specialized hardware and software
- Multiple processors in a shared memory configuration
- Required modest changes to software (vectorization)
The attack of the killer micros

- The **Caltech Cosmic Cube** developed by Charles Seitz and Geoffrey Fox in 1981
- 64 Intel 8086/8087 processors
- 128kB of memory per processor
- 6-dimensional hypercube network

Launched the “attack of the killer micros”
Eugene Brooks, SC’90

The cosmic cube, Charles Seitz
Communications of the ACM, Vol 28, number 1 January 1985, p. 22

http://calteches.library.caltech.edu/3419/1/Cubism.pdf
It took a while, but MPPs came to dominate supercomputing

- Parallel computers with large numbers of microprocessors
- High speed, low latency, scalable interconnection networks
- Lots of custom hardware to support scalability
- Required massive changes to software (parallelization)

![Graph showing peak GFLOPS for different machines]

- iPSC\(860(128)\) 1990.
- TMC CM5\((1024)\) 1992
- Paragon XPS 1993

Paragon XPS-140 at Sandia National labs in Albuquerque NM
The cost advantage of mass market COTS

- MPPs using Mass market Commercial off the shelf (COTS) microprocessors and standard memory and I/O components
- Decreased hardware and software costs makes huge systems affordable
The MPP future looked bright … but then clusters took over

- A cluster is a collection of connected, independent computers that work in unison to solve a problem.
- Nothing is custom … motivated users could build cluster on their own
- First clusters appeared in the late 80’s (Stacks of “SPARC pizza boxes”)
- The Intel Pentium Pro in 1995 coupled with Linux made them competitive.
  - NASA Goddard’s Beowulf cluster demonstrated publically that high visibility science could be done on clusters.
- Clusters made it easier to bring the benefits due to Moores’s law into working supercomputers
Top 500 list: System Architecture

*Constellation: A cluster for which the number of processors on a node is greater than the number of nodes in the cluster. I’ve never seen anyone use this term outside of the top500 list.

Source: http://s.top500.org/static/lists/2013/06/TOP500_201306_Poster.pdf

*Constellation: A cluster for which the number of processors on a node is greater than the number of nodes in the cluster. I’ve never seen anyone use this term outside of the top500 list.
Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
    - Message passing details
    - Collective operations
- Closing comments
MPI (1992-today)

- The message passing interface (MPI) is a standard library
- MPI Forum first met April 1992,
  - MPI 1.0 in June 1994
  - MPI 2.0 in July 1997
  - MPI 3.0 in September 2012
- Hardware-portable, multi-language communication library
- Enabled billions of dollars of applications
- Work on MPI 3.1 and 4.0 is in progress.

MPI Forum, March 2008, Chicago
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv) {
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf("Hello from process %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
# Initializing and finalizing MPI

```c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

**int MPI_Init (int* argc, char* argv[])**

- Initializes the MPI library ... called before any other MPI functions.
- argc and argv are the command line args passed from main()

**int MPI_Finalize (void)**

- Frees memory allocated by the MPI library ... close every MPI program with a call to MPI_Finalize
int MPI_Comm_size (MPI_Comm comm, int* size)

- **MPI_Comm**, an *opaque data type*, a communication context. Default context: MPI_COMM_WORLD (all processes)
- **MPI_Comm_size** returns the number of processes in the process group associated with the communicator

```c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

**Communicators** consist of two parts, a *context* and a *process group*.

The communicator lets me control how groups of messages interact.

The communicator lets me write modular SW ... i.e. I can give a library module its own communicator and know that it’s messages can’t collide with messages originating from outside the module.
Which process “am I” (the rank)

```c
#include <stdio.h>
#include <mpi.h>

int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- **MPI_Comm**, an opaque data type, a communication context. Default context: MPI_COMM_WORLD (all processes)
- **MPI_Comm_rank** An integer ranging from 0 to “(num of procs)-1”

Note that other than init() and finalize(), every MPI function has a communicator.

This makes sense .. You need a context and group of processes that the MPI functions impact … and those come from the communicator.
Running the program

```c
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
            rank, size );
    MPI_Finalize();
    return 0;
}
```

- On a 4 node cluster with MPIch2, I’d run this program (hello) as:
  - `mpicc hello.c –o hello`
  - `mpiexec –n 4 –f hostf hello`
  - Hello from process 1 of 4
  - Hello from process 2 of 4
  - Hello from process 0 of 4
  - Hello from process 3 of 4
- Where “hostf” is a file with the names of the cluster nodes, one to a line.
Sending and Receiving Data

- **MPI_Send** performs a blocking send of the specified data ("count" copies of type "datatype," stored in "buf") to the specified destination (rank "dest" within communicator "comm"), with message ID "tag"

- **MPI_Recv** performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in "status"

By "blocking" we mean the functions return as soon as the buffer, "buf", can be safely used.
The data in a message to send or receive is described by a triple:

- \((\text{address}, \text{count}, \text{datatype})\)

An MPI datatype is recursively defined as:

- Predefined, simple data type from the language (e.g., MPI_DOUBLE)
- Complex data types (contiguous blocks or even custom types).

E.g. … A particle’s state is defined by its 3 coordinates and 3 velocities:

```c
MPI_Datatype PART;
MPI_Type_contiguous( 6, MPI_DOUBLE, &PART );
MPI_Type_commit( &PART );
```

You can use this data type in MPI functions, for example, to send data for a single particle:

```c
MPI_Send (buff, 1, PART, Dest, tag, MPI_COMM_WORLD);
```
Receiving the right message

- The receiving process identifies messages with the double `:(source, tag)`.
- Where:
  - Source is the rank of the sending process.
  - Tag is a user-defined integer to help the receiver keep track of different messages from a single source.

```c
MPI_Recv (buff, 1, PART, Src, tag, MPI_COMM_WORLD, &status);
```

- Can relax tag checking by specifying `MPI_ANY_TAG` as the tag in a receive.
- Can relax source checking by specifying `MPI_ANY_SOURCE`

```c
MPI_Recv (buff, 1, PART, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

- This is a useful way to insert race conditions into an MPI program.
How do people use MPI?
The SPMD Design Pattern

A sequential program working on a data set

• A single program working on a decomposed data set.

• Use Node ID and numb of nodes to split up work between processes

• Coordination by passing messages.

Replicate the program.
Add glue code
Break up the data
#include “mpi.h”
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    printf( “Received %d\n”, buf );
  }
  MPI_Finalize();
  return 0;
}
Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
    - Collective operations
- Closing comments
Buffers

- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance

- When you send data, where does it go? One possibility is:

  Process 0
  
  Process 1

User data → Local buffer → the network → Local buffer → User data

8/20/2013

Derived from: Bill Gropp, UIUC
Blocking Send-Receive Timing Diagram
(Receive before Send)

send side

receive side

MPI_Send: T1

MPI_Send returns T2

Local buffer can be reused

T0: MPI_Recv
Once receive is called @ T0, Local buffer unavailable to user

T3: Transfer Complete

T4: MPI_Recv returns
Local buffer filled and available to user

It is important to post the receive before sending, for highest performance.
Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- This code could deadlock ... it depends on the availability of system buffers in which to store the data sent until it can be received

Slide source: based on slides from Bill Gropp, UIUC
Some Solutions to the “deadlock” Problem

- Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Supply receive buffer at same time as send:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendrecv(1)</td>
<td>Sendrecv(0)</td>
</tr>
</tbody>
</table>
More Solutions to the “unsafe” Problem

- Supply a sufficiently large buffer in the send function

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bsend(1)</td>
<td>Bsend(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>
Non-Blocking Communication

- Non-blocking operations return immediately and pass “request handles” that can be waited on and queried
  - `MPI_ISEND( start, count, datatype, dest, tag, comm, request )`
  - `MPI_Irecv( start, count, datatype, src, tag, comm, request )`
  - `MPI_WAIT( request, status )`

- One can also test without waiting using `MPI_TEST`
  - `MPI_TEST( request, flag, status )`

- Anywhere you use `MPI_Send` or `MPI_Recv`, you can use the pair of `MPI_Isend/MPI_Wait` or `MPI_Irecv/MPI_Wait`

Non-blocking operations are extremely important ... they allow you to overlap computation and communication.
Non-Blocking Send-Receive Diagram

- **send side**
  - MPI_Isend (T2)
  - MPI_Isend returns (T3)
  - Sender completes (T5)
  - MPI_Wait (T6)
  - MPI_Wait returns (T9)

- **receive side**
  - T0: MPI_Irecv
  - T1: MPI_Irecv Returns
  - buffer unavailable to user
  - T4: MPI_Wait called
  - T7: transfer finishes
  - T8: MPI_Wait returns
  - receive buffer filled and available to the user

- buffer unavailable to user

- buffer available to user
Example: shift messages around a ring (part 1 of 2)

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int num, rank, size, tag, next, from;
    MPI_Status status1, status2;
    MPI_Request req1, req2;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    tag = 201;
    next = (rank+1) % size;
    from = (rank + size - 1) % size;
    if (rank == 0) {
        printf("Enter the number of times around the ring: ");
        scanf("%d", &num);
        printf("Process %d sending %d to %d\n", rank, num, next);
        MPI_Isend(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD,&req1);
        MPI_Wait(&req1, &status1);
    }

    printf("Process %d sending %d to %d\n", rank, num, next);
    MPI_Isend(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD,&req1);
    MPI_Wait(&req1, &status1);
}
```
do {
    MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
    MPI_Wait(&req2, &status2);
    printf("Process %d received %d from process %d\n", rank, num, from);

    if (rank == 0) {
        num--;
        printf("Process 0 decremented number\n");
    }

    printf("Process %d sending %d to %d\n", rank, num, next);
    MPI_Isend(&num, 1, MPI_INT, next, tag, MPI_COMM_WORLD, &req1);
    MPI_Wait(&req1, &status1);
} while (num != 0);

if (rank == 0) {
    MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
    MPI_Wait(&req2, &status2);
}

MPI_Finalize();
return 0;
Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments
Reduction

```
int MPI_Reduce (void* sendbuf,
        void* recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op,
        int root, MPI_Comm comm)
```

- **MPI_Reduce** performs specified reduction operation on specified data from all processes in communicator, places result in process “root” only.
- **MPI_Allreduce** places result in all processes (avoid unless necessary)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUM</td>
<td>Summation</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum value</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum value and location</td>
</tr>
<tr>
<td>MPI_MAX</td>
<td>Maximum value</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum value and location</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BAND</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive OR</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive OR</td>
</tr>
<tr>
<td>User-defined</td>
<td>It is possible to define new reduction operations</td>
</tr>
</tbody>
</table>


#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);

    for (i=my_id; i<num_steps; i=i+numprocs)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, 
               MPI_COMM_WORLD);
}
MPI Pi program performance

**Pi program in MPI**

```c
#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    for (i=my_id; i<numprocs; i=i+numprocs)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
}
```

<table>
<thead>
<tr>
<th>Thread or procs</th>
<th>OpenMP SPMD critical</th>
<th>OpenMP PI Loop</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.85</td>
<td>0.43</td>
<td>0.84</td>
</tr>
<tr>
<td>2</td>
<td>0.48</td>
<td>0.23</td>
<td>0.48</td>
</tr>
<tr>
<td>3</td>
<td>0.47</td>
<td>0.23</td>
<td>0.46</td>
</tr>
<tr>
<td>4</td>
<td>0.46</td>
<td>0.23</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Note: OMP loop used a Blocked loop distribution. The others used a cyclic distribution. Serial .. 0.43.

*Intel compiler (icpc) with –O3 on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.
Many MPI applications have few (if any) sends and receives. They use a design pattern called **Bulk Synchronous Processing**.

- Uses the Single Program Multiple Data pattern
- Each process maintains a local view of the global data
- A problem broken down into phases each composed of two subphases:
  - Compute on local view of data
  - Communicate to update global view on all processes (collective communication).
- Continue phases until complete
Collective communications: called by all processes in the group to create a global result and share with all participating processes.

- Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv

Notes:
- Allreduce, Reduce, Reduce_scatter, and Scan use the same set of built-in or user-defined combiner functions.
- Routines with the “All” prefix deliver results to all participating processes.
- Routines with the “v” suffix allow chunks to have different sizes.

Global synchronization is available in MPI
- MPI_BARRIER( comm )

Blocks until all processes in the group of the communicator comm call it.
Collective Data Movement

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

Broadcast

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
</tbody>
</table>

Scatter

<table>
<thead>
<tr>
<th>P0</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>B</td>
</tr>
<tr>
<td>P2</td>
<td>C</td>
</tr>
<tr>
<td>P3</td>
<td>D</td>
</tr>
</tbody>
</table>

Gather
More Collective Data Movement

For Allgather:

- P0: A
- P1: B
- P2: C
- P3: D

After Allgather:

- P0: A
- P1: B
- P2: C
- P3: D

For Alltoall:

- P0: A0, A1, A2, A3
- P1: B0, B1, B2, B3
- P2: C0, C1, C2, C3
- P3: D0, D1, D2, D3

After Alltoall:

- P0: A0, B0, C0, D0
- P1: A1, B1, C1, D1
- P2: A2, B2, C2, D2
- P3: A3, B3, C3, D3
Collective Computation

P0  A
P1  B
P2  C
P3  D

Reduce

P0  A
P1  B
P2  C
P3  D

Scan

ABCD

A
AB
ABC
ABCD
Outline

- Distributed memory systems: the evolution of HPC hardware
- Programming distributed memory systems with MPI
  - MPI introduction and core elements
  - Message passing details
  - Collective operations
- Closing comments
MPI topics we did Not Cover

- Topologies: map a communicator onto, say, a 3D Cartesian processor grid
  - Implementation can provide ideal logical to physical mapping
- Rich set of I/O functions: individual, collective, blocking and non-blocking
  - Collective I/O can lead to many small requests being merged for more efficient I/O
- One-sided communication: puts and gets with various synchronization schemes
  - Implementations not well-optimized and rarely used
  - Redesign of interface is underway
- Task creation and destruction: change number of tasks during a run
  - Few implementations available
There are over 330 functions in the MPI spec, but most programs only use a small subset:

- **Point-to-point communication**
  - `MPI_Irecv`, `MPI_Isend`, `MPI_Wait`, `MPI_Send`, `MPI_Recv`
- **Startup**
  - `MPI_Init`, `MPI_Finalize`
- **Information on the processes**
  - `MPI_Comm_rank`, `MPI_Comm_size`
- **Collective communication**
  - `MPI_Allreduce`, `MPI_Bcast`, `MPI_Allgather`
Isn’t message passing much harder than multithreading?

Extra work upfront, but easier optimization and debugging means overall, less time to solution

---

Proving that a shared address space program using semaphores is race free is an NP-complete problem*

---

MPI References

- The Standard itself:
  - at [http://www.mpi-forum.org](http://www.mpi-forum.org)
  - All MPI official releases, in both postscript and HTML

- Other information on Web:
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages
**Books on MPI**

- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.

Slide source: Bill Gropp, ANL
The key constructs of MPI

- MPI_Init() and MPI_Finalize()
- MPI_Comm_rank() and MPI_Comm_size()
- MPI_Send() and MPI_Recv()
- MPI_Isend(), MPI_Irecv(), and MPI_Wait()
- MPI_Bcast(), MPI_Reduce(), MPI_Gather(), and MPI_Scatter()
- MPI_Barrier()

To do: I need a page for each one of these similar to the one I have now for MPI_send and MPI_Recv
### Blocking Send and Receive

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send</td>
<td>Performs a blocking send of the specified data (&quot;count&quot; copies of type &quot;datatype,&quot; stored in &quot;buf&quot;) to the specified destination (rank &quot;dest&quot; within communicator &quot;comm&quot;), with message ID &quot;tag&quot;</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in &quot;status&quot;</td>
</tr>
</tbody>
</table>

By "blocking" we mean the functions return as soon as the buffer, "buf", can be safely used.
Non-Blocking Communication

- Non-blocking operations return immediately and pass “request handles” that can be waited on and queried
  - `MPI_ISEND( start, count, datatype, dest, tag, comm, request )`
  - `MPI_IRECV( start, count, datatype, src, tag, comm, request )`
  - `MPI_WAIT( request, status )`
- One can also test without waiting using `MPI_TEST`
  - `MPI_TEST( request, flag, status )`
- Anywhere you use `MPI_Send` or `MPI_Recv`, you can use the pair of `MPI_Isend/MPI_Wait` or `MPI_Irecv/MPI_Wait`

Non-blocking operations are extremely important … they allow you to overlap computation and communication.
Launching and closing MPI

- These functions “bracket” every MPI program

```c
int MPI_Init (int* argc, char* argv[])

- Initializes the MPI library … called before any other MPI functions.
- argc and argv are the command line args passed from main()

int MPI_Finalize (void)

- Frees memory allocated by the MPI library … close every MPI program with a call to MPI_Finalize
```
Understanding the process group

- SPMD pattern: use the ID of each process and the size of the process group to choose the data manipulated or the branching through the program

```c
int MPI_Comm_size (MPI_Comm comm, int* size)
- MPI_Comm, an opaque data type, a communication context. Default context: MPI_COMM_WORLD (all processes)
- MPI_Comm_size returns the number of processes in the process group associated with the communicator

int MPI_Comm_rank (MPI_Comm comm, int* rank)
- MPI_Comm, an opaque data type, a communication context. Default context: MPI_COMM_WORLD (all processes)
- MPI_Comm_rank An integer ranging from 0 to “(num of procs)-1”
```